Vehicle Routing Algorithm
Algorithms and Data Structures
Coursework 1

Sam Dixon
40056761
SET09117

1 Introduction

Vehicle Routing The Vehicle Routing Problem has always been a difficult computational scenario to solve. In its simplest form, the problem is as follows: a depot must deliver to a group of customers, each requiring a varying number of goods, using vehicles that have limited cargo space. While it may appear a simple task from the outset, it quickly becomes clear that the main difficulty arises from the sheer number of possible combinations of potential routes. For example, consider a scenario of only ten customers will have up to 3,628,800 possible route combinations. It is apparent that a 'brute force' methodology will not solve this problem in a timely manner, and that a heuristic method must be applied.

Existing Solutions One solution to the Vehicle Routing Problem exists in the form of the well-established Clarke Wright Algorithm, which is discussed at length in their own paper: 'Scheduling of Vehicles from a Central Depot to a Number of Delivery Points' [Clarke and Wright 1962]. This Algorithm works by creating pairs of customers, and calculating the distance saved by visiting both customers together, rather than individually. Finally, the algorithm attempts to join the pairs with the highest savings together into coherent routes. The algorithm breaks the problem into three key aspects. Customers, Vehicles and Routes. An example of a Clarke Wright solution can be seen in figure 1.

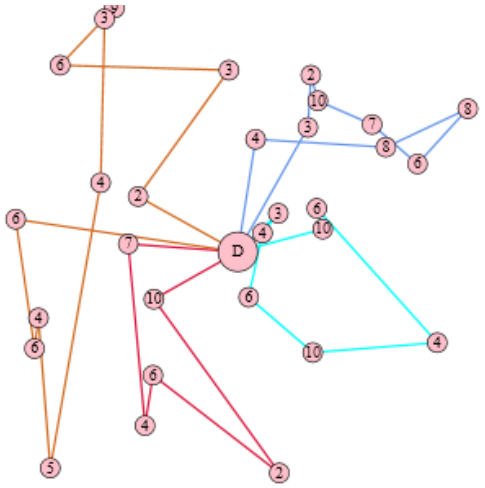

Figure 1: *Clarke Wright solution of a of thirty Customer scenario.*

Customers In this specification, a customer requires N number of goods to be delivered from a depot. It is assumed that all customers are served from a single depot, each customer must receive the total number of required goods and no customer should be left unserved.

Vehicles Each vehicle has a total goods capacity and it is assumed that all vehicles used by a depot share the same total capacity. It is also assumed that each vehicle begins and ends its route at the depot.

Routes In essence, routes are simply a list of customers, in the order that they to be visited. In order to successfully complete a scenario, several routes may have to be generated to ensure that each customer is visited.

Presented Solution The algorithm presented in this report (hence forth referred to as The Simple Route Algorithm) is a personal interpretation of The Clarke Wright Algorithm. The results produced by The Simple Route algorithm differ from Clarke Wright, mainly in number of routes and appearance, whereas the overall cost remains very similar.

Simple Route Functionality The Simple Route Algorithm works similarly to The Clarke Wright. Routes are generated in parallel - if a pair of customers cannot be merged with an existing route, it becomes its own route which future pairs will attempt to join to. The primary difference between this algorithm and a parallel Clarke Wright is that full routes are never merged together.
The reasons for this are twofold - firstly, merging two routes together may reduce the total number of routes needed to solve a scenario, but this can also leave customers in routes by themselves.
The second reason for not merging is that it can often cause routes to form U-turns, or to cross over themselves unnecessarily. In a real life scenario, a company would not utilise a route that needlessly intersects, thus making the solutions provided by The Simple Route Algorithm a more appealing choice. An example of a Simple Route solution is presented in figure 2.

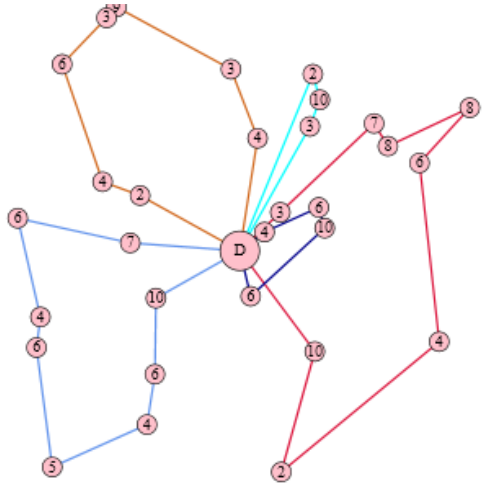

Figure 2: *Simple Route solution of a of thirty Customer scenario.*

Simple Route Limitations The Simple Route Algorithm produces solutions that consist of more routes than a Clarke Wright Algorithm would. These paths are also of lower cargo capacity than either the sequential or parallel Clarke Wright. This means that if a company were to employ a Simple Route solution, they would require a larger fleet of vehicles to deliver to all customers, simultaneously.

2 Method

Algorithm Testing The solutions produced by the Simple Route Algorithm were converted to .csv files, and tested using the provided verifier.jar to ensure that each customer was visited and that no route ran over its goods capacity. Visual representations of the solutions in .svg format were also used to ensure that the routes created would be practical to use in the context of a real life situation.
In-line testing was also used to ensure the validity of the provided code. Printing out the cargo needed for each route and customers yet to be visited made debugging of faulty code a simpler task.

Algorithm Comparisons In order to compare the quality of solutions generated by The Simple Route Algorithm, test data had to be created. A basic algorithm that allocated each customer to its own route was used to produce the most expensive solution for each scenario. Data was also derived for the provided Clarke Wright solutions.
By comparing the results of all three algorithms, it is clear to see that in regards to overall cost, The Simple Route Algorithm performs just as well as the Clarke Wright and is significantly more cost-effective than The Single Customer solution. All data is visible in table 1.

Testing Process In order to generate completion times for The Simple Route Algorithm, each scenario was performed one-hundred times and the results were averaged. This was done in order to reduce external factors having an effect on the results of the experiment. All tests were completed on a 2.10GHz i3-2310M CPU.

Data Inconsistencies Comparison of the various algorithms results has revealed some inconsistencies in the test results. As one can seen in figure 5, the results obviously fluctuate, especially in the scenarios consisting of fewer customers. These differences have occurred due to a lack of different scenarios from each number of customers. If several problems had been provided, each with the same number of customers, the results could have been averaged and thus more accurate data could have been collected.

3 Results

Number of Customers	Single Customer Cost	Clarke Wright Cost	Simple Route Cost	Routes Required	Time (Milliseconds)
10	3781.37	1576.53	1587.06	3	0.17
20	7930.54	2698.80	2679.09	4	0.3
30	10302.35	3781.33	3181.68	5	0.66
40	15775.47	3727.15	3606.24	6	1.48
50	20072.57	5411.73	5174.34	9	2.99
60	23193.74	6015.10	5941.35	11	4.89
70	25924.87	5898.03	5107.64	9	7.38
80	28756.68	6882.72	7069.06	12	10.85
90	34580.28	8441.57	7200.53	13	15.66
100	38703.90	Not Provided	8061.22	14	21.15
200	76928.55	Not Provided	13409.73	25	162.85
300	116480.94	Not Provided	17863.18	34	547.27
400	153608.74	Not Provided	23106.65	48	1279.6
500	188739.51	Not Provided	28351.77	59	2500.01
600	228961.57	Not Provided	33861.05	73	4287.25
700	266674.71	Not Provided	37652.28	80	6726.07
800	311675.49	Not Provided	42334.97	90	10083.94
900	347699.30	Not Provided	49761.74	108	14364.07
1000	390783.40	Not Provided	51681.50	110	19555.05

Table 1: *Results of all tests performed.*

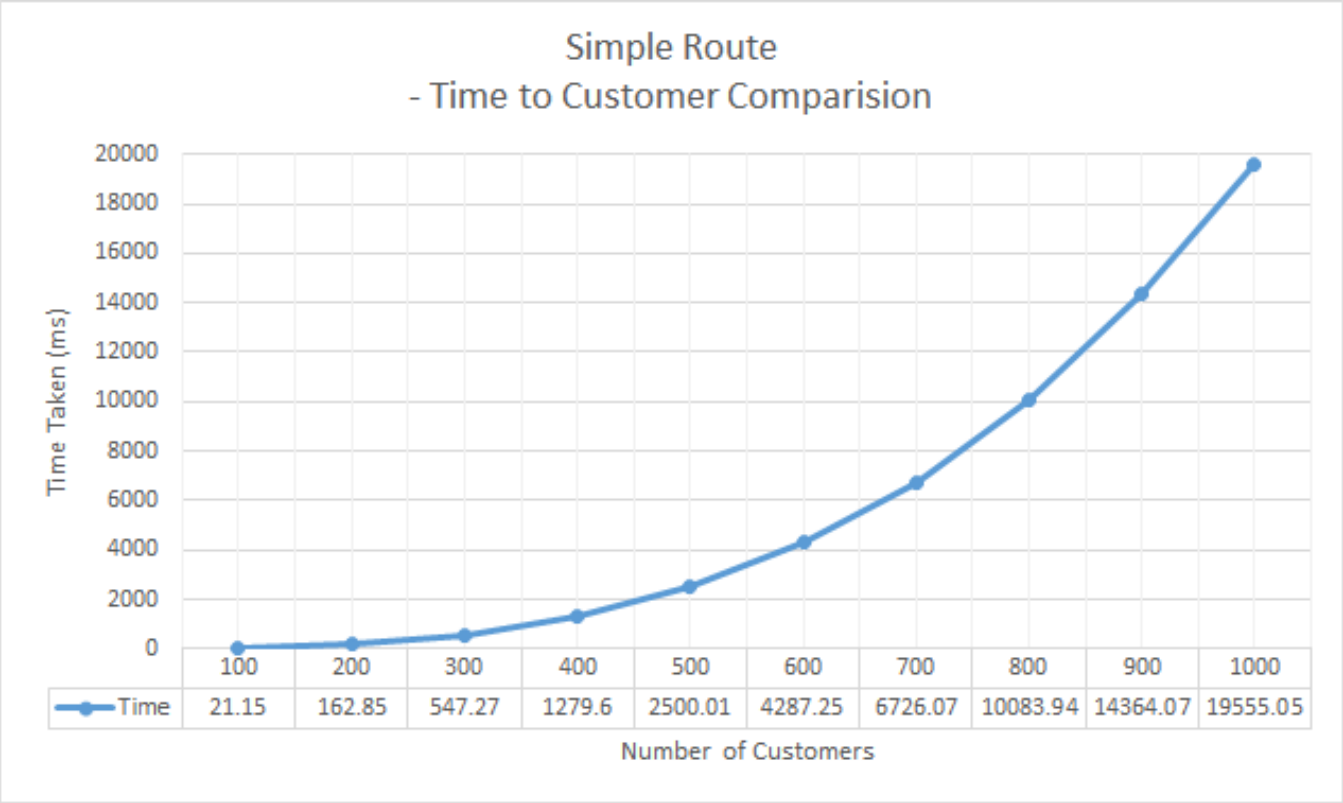

Figure 3: ***Simple Route Time Results*** *- Average completion time of The Simple Route Algorithm compared to number of customers.*

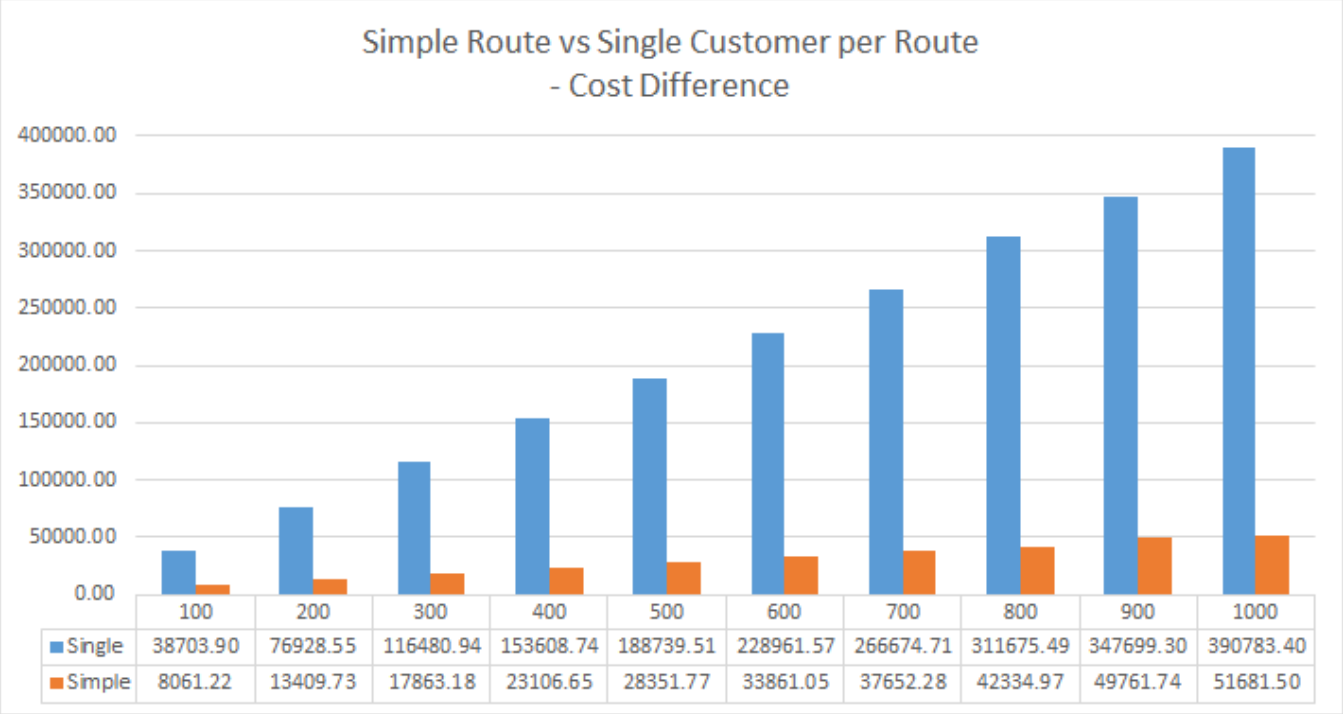

Figure 4: ***Simple Route vs Single Customer per Route*** *- The difference in cost when comparing The Simple Route against The Single Customer per Route Algorithm.*

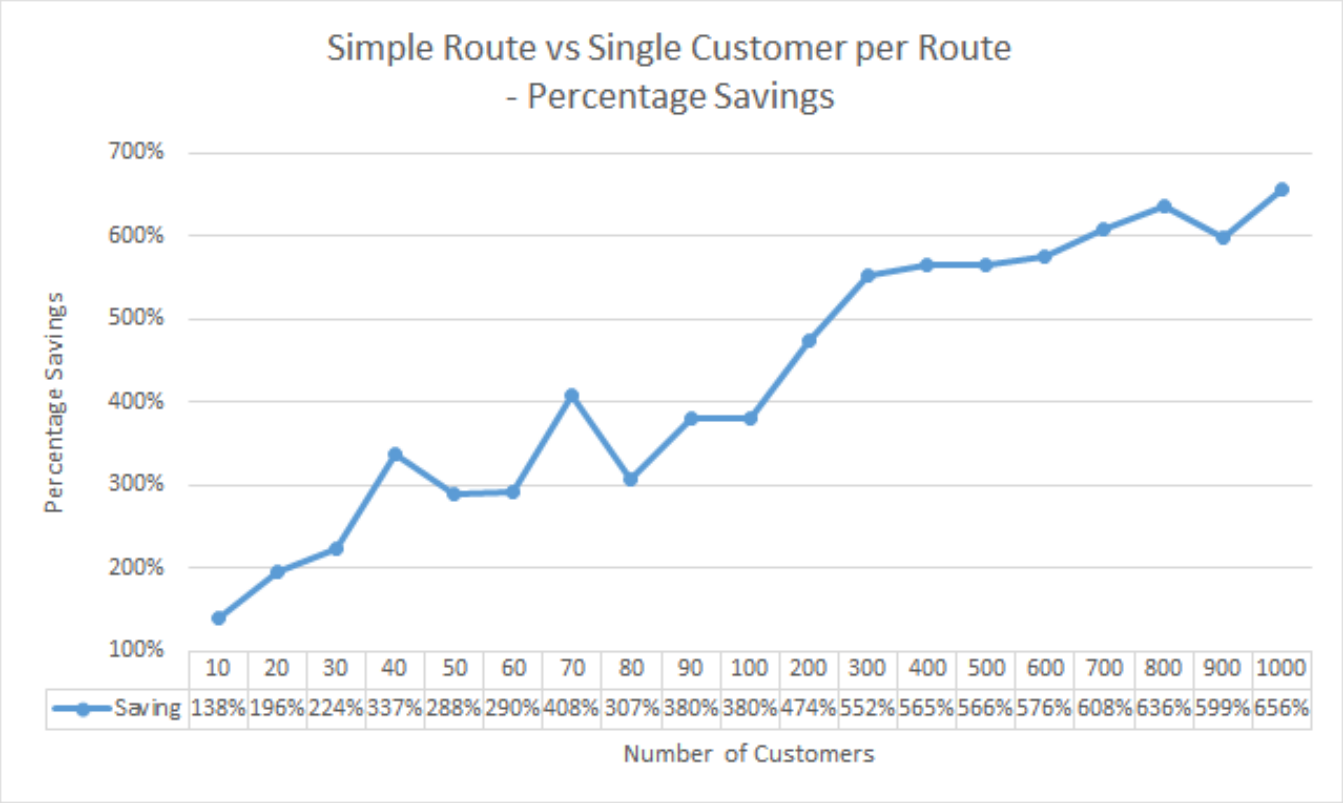

Figure 5: ***Simple Route Percentile Saving*** *- The percentage difference between using The Simple Route Algorithm compared to The Single Customer per Route Algorithm.*

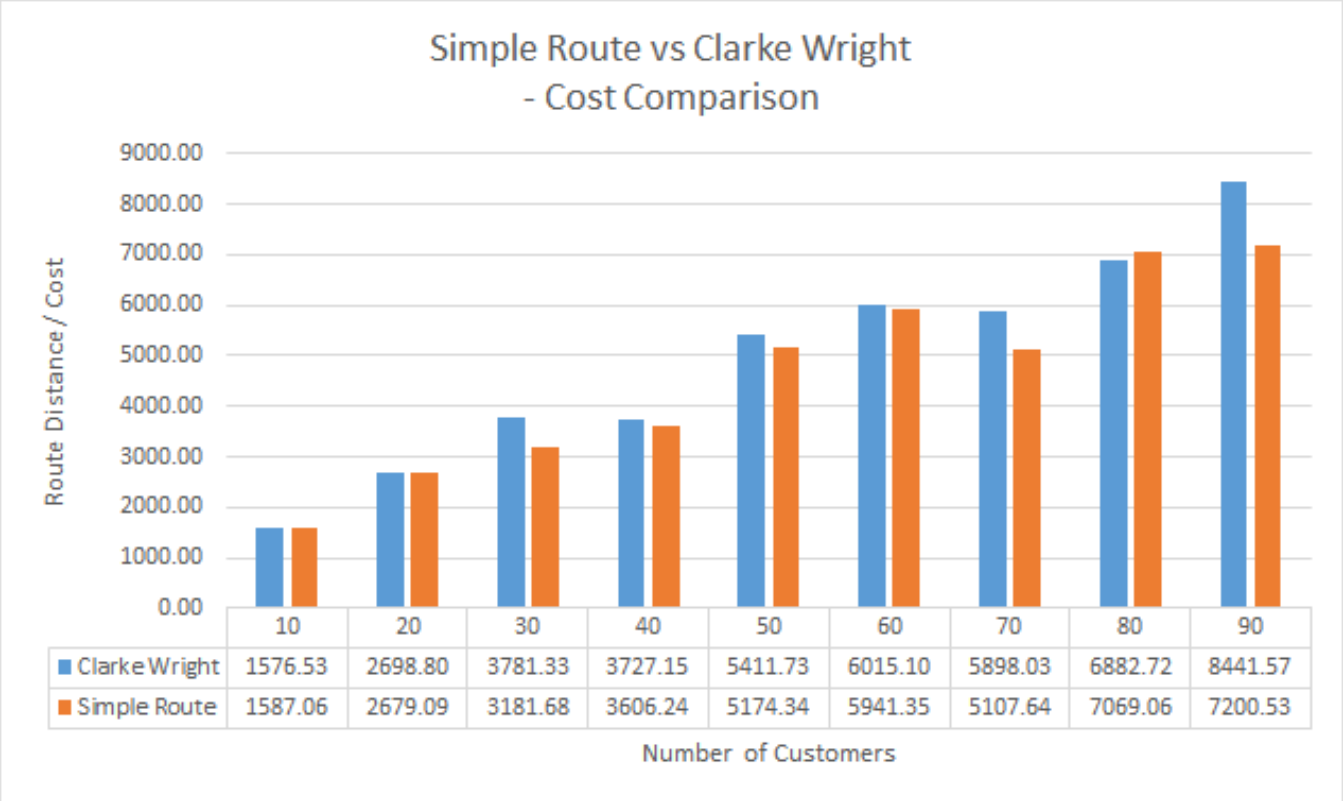

Figure 6: ***Simple Route vs Clarke Wright*** *- The difference in cost when comparing the Simple Route against the Clarke Wright Algorithm*

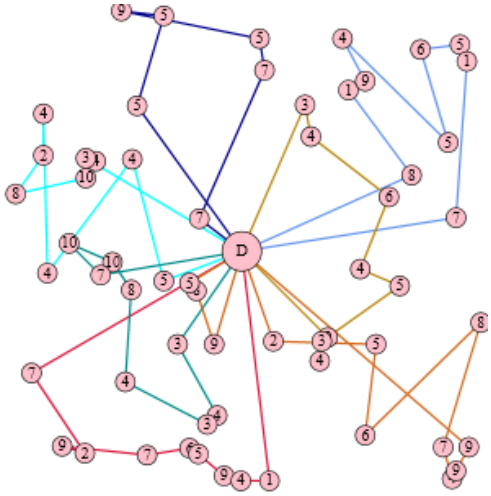

Figure 7: ***Clarke Wright solution of a problem consisting of sixty Customers.***

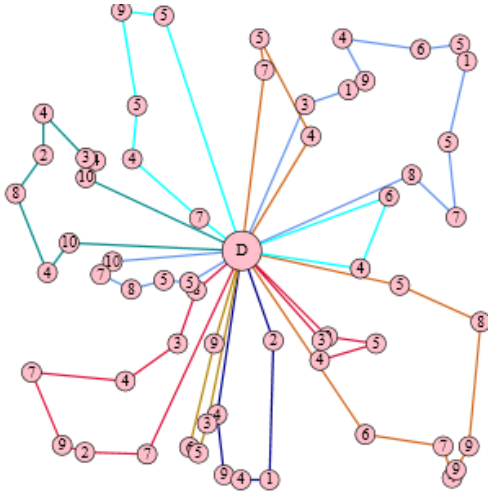

Figure 8: ***Simple Route solution of a problem consisting of sixty Customers.***

4 Conclusion

Summary of Results As can be seen in figure 4, The Simple Route performs more efficiently than The Single Customer Algorithm in regards to overall cost and it is also extremely close to Clarke Wright, in this respect, see figure 6.
The Simple Route Algorithm is arguably more efficient than Clarke Wright in terms of its visual practicality. The routes generated by The Simple Route have fewer or no U-turns and cross overs, providing solutions that would be more practical in a real life situation. (See figures 7 and 8)
The primary goal of The Simple Route Algorithm was to produce solutions that did not generate routes in which customers were on their own. Using the data provided, this objective was achieved 100% of the time.
The only occasion in which The Simple Route Algorithm would fail this goal would be if a customer's goods requirement was equal to, or very close to the vehicle's total cargo capacity.
The reason to prevent a customer from being on their own was to produce more practical routes, as it is very unlikely that in a real life situation a company would plan for one vehicle to leave a depot to deliver an order to a single customer.
The discovery of the lack of intersections was initially an unplanned side effect. However, after researching the benefits of this, it was implemented fully into The Simple Route Algorithm.

Performance of Assessment During the original implementation of The Simple Route Algorithm, it performed very poorly, mainly due to the amount of time it took to calculate a solution - initially taking up to fifty-one minutes to generate a result for an eight-hundred customer scenario. By timing each subroutine of The Simple Route Algorithm, it was found that the most expensive process was the elimination of duplicate pairs of customers. By re-factoring the customer pairing method so that duplicate customers could not be created, it became possible to reduce the time from fifty-one minutes to approximately ten seconds.
The reduction of crossovers and U-turns was caused by omitting the route-merging method of The Clarke Wright Algorithm. Not only does this create more practical routes, but it also reduces the amount of code necessary to implement a solution to The Vehicle Routing Problem, as The Simple Route Algorithm no longer has to check for lone customers or accidental loops in the journey.
While the results from testing prove the efficiency and validity of The Simple Route Algorithm, more accurate results could be produced if more test data was provided.
The Simple Route Algorithm presented in this report is a valid solution The Vehicle Routing problem, resulting in a similar overall cost to The Clarke Wright Algorithm, but with the added benefit of more practical routes that could be utilised in real life situations.

5 Appendix

5.1 Execute.java

```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class Execute {
5
6    public static void main(String[] args) throws Exception {
7       single("rand00010prob");
8       benchmark(1);
9    }
10
11   // calculates a solution for the provided problem
12   private static void single(String prob) throws Exception {
13      VRProblem p = new VRProblem("test_data/" + prob + ".csv");
14      VRSolution s = new VRSolution(p);
15
16      // start time
17      double startTime = System.currentTimeMillis();
18      // execute algorithm
19      s.simpleRouteSolution();
20      // stop time
21      double endTime = System.currentTimeMillis();
22
23      // creates a .csv of the solution
24      s.writeOut("solutions/" + prob + "Solution.csv");
25      // create .svg of the problem and the solution
26      s.writeSVG("solutions/" + prob + ".svg", "solutions/" + prob + ↩
          "Solution.svg");
27
28      // name of problem
29      System.out.println(prob);
30      // verify solution is valid
31      System.out.println(s.verify() + " solution");
32      // returns number of routes needed to complete the problem
33      System.out.println("routes: " + s.numberOfRoutes());
34      // cost of problem
35      System.out.println("cost: " + s.solnCost());
36      // time to complete calculation
37      System.out.println("time: " + (endTime − startTime));
38   }
```

```
39
40    // run each problem n times to create averages
41    private static void benchmark(int n) throws Exception {
42        // list of all problems
43        String[] problems = new String[] { "rand00010prob", "rand00020prob", "rand00030prob", "rand00040prob",
44            "rand00050prob", "rand00060prob", "rand00070prob", "rand00080prob", "rand00090prob", "rand00100prob",
45            "rand00200prob", "rand00300prob", "rand00400prob", "rand00500prob", "rand00600prob", "rand00700prob",
46            "rand00800prob", "rand00900prob", "rand01000prob" };
47
48        // loop through every problem
49        for (String prob : problems) {
50            List<Double> times = new ArrayList<>();
51            VRProblem p = new VRProblem("test_data/" + prob + ".csv");
52            VRSolution s = new VRSolution(p);
53
54            // run each problem n times
55            for (int i = 0; i < n; i++) {
56                double startTime = System.currentTimeMillis();
57                s.simpleRouteSolution();
58                double endTime = System.currentTimeMillis();
59
60                // creates a .csv of the solution
61                s.writeOut("solutions/" + prob + "Solution.csv");
62                // create .svg of the problem and the solution
63                s.writeSVG("solutions/" + prob + ".svg", "solutions/" + prob + "Solution.svg");
64
65                // add time to list
66                times.add(endTime - startTime);
67            }
68            // name of problem
69            System.out.println(prob);
70            // verify solution is valid
71            System.out.println(s.verify() + " solution");
72            // returns number of routes needed to complete the problem
73            System.out.println("routes: " + s.numberOfRoutes());
74            // cost of problem
75            System.out.println("cost: " + s.solnCost());
76            System.out.println(times);
77        }
78    }
79
80 }
```

5.2 VRSolution.java

Lines 23 to 27

```
1    // Returns solution generated by Simple Route algorithm
2    public void simpleRouteSolution() {
3        SimpleRouteAlg cw = new SimpleRouteAlg();
4        this.soln = cw.solution(prob);
5    }
```

5.3 Route.java

```
1  import java.util.ArrayList;
2
3  //class for storage of customers in the same route, and calculating cost/saving
4  public class Route implements Comparable<Route> {
5
6      // variable declaration
7      private ArrayList<Customer> _customerList;
8      private static Customer _depot;
9      private double _saving;
10     private int _requirment;
11
12     // constructor
13     public Route() {
14         _customerList = new ArrayList<Customer>();
15         _saving = 0;
16         _requirment = 0;
17     }
18
19     // calculate distance saving of a route
20     public void calcSaving() {
21         double depotCost = 0, routeCost = 0;
22
23         // reset saving each time calculation is performed
24         _saving = 0;
25         // loop through each customer in the route
26         for (Customer c : this._customerList) {
27             // calculate distance travelled if only that customer is visited
28             depotCost += _depot.distance(c);
29         }
30
31         // loop through each customer
32         for (int i = 0; i < this._customerList.size() - 1; i++) {
33             // calculate distance between each customer and its neighbour
34             routeCost += _customerList.get(i).distance(_customerList.get(i + 1));
35         }
36
37         // saving is difference between single customer cost and route cost
38         _saving = depotCost - routeCost;
39     }
40
41     // customer comparable used for sorting collections of routes by saving
42     @Override
43     public int compareTo(Route compareRoute) {
44         if (this._saving < compareRoute._saving) {
45             return 1;
46         } else {
47             return -1;
48         }
49     }
50
51     // append a customer to either end of the array
52     public void addToEnd(Customer cust) {
53         _customerList.add(cust);
54         _requirment += cust.c;
55     }
56
57     public void addToStart(Customer cust) {
58         _customerList.add(0, cust);
59         _requirment += cust.c;
60     }
61
62     // setters and getters
63     public static void set_depot(Customer _depot) {
64         Route._depot = _depot;
65     }
66
67     public Customer getLastCustomer() {
68         return _customerList.get(_customerList.size() - 1);
69     }
70
71     public Customer getFirstCustomer() {
72         return _customerList.get(0);
73     }
74
75     public ArrayList<Customer> get_customerList() {
76         return _customerList;
77     }
78
79     public double get_saving() {
80         return _saving;
81     }
82
83     public int get_requirment() {
84         return _requirment;
85     }
86
87 }
```

5.4 simpleRouteAlg.java

```
import java.util.ArrayList;

//returns solution to a problem using SRAlg
public class SimpleRouteAlg {
    // variables
    private VRProblem _prob;
    private ArrayList<Route> _pairs;

    // create routes using Simple Route Algorithm
    public ArrayList<ArrayList<Customer>> solution(VRProblem prob) {
        // define variables
        ArrayList<ArrayList<Customer>> sol = new ArrayList<ArrayList<Customer>>();
        this._prob = prob;

        // create pairs
        createPairs();
        // sort pairs in descending order of savings
        _pairs.sort(null);
        // build final routes and add them to solution
        for (Route route : buildRoutes()) {
            sol.add(route.get_customerList());
        }
        // return solution
        return sol;
    }

    // creates routes for every possible pair of customers
    private void createPairs() {
        // define ArrayList for storage of pairs
        this._pairs = new ArrayList<Route>();

        // set the depot of all routes so costs/savings can be calculated later
        Route.set_depot(_prob.depot);
        // loop through every customer
        for (int i = 0; i < _prob.customers.size(); i++) {
            // loop through every customer
            for (int j = i; j < _prob.customers.size(); j++) {
                // if the pair is not the same customer twice
                if (i != j) {
                    // create a new route
                    Route route = new Route();
                    // add each customer to route
                    route.addToEnd(_prob.customers.get(i));
                    route.addToEnd(_prob.customers.get(j));
                    // Calculate the saving
                    route.calcSaving();
                    // add route to ArrayList
                    _pairs.add(route);
                }
            }
        }
    }

    // combine pairs to create full routes
    private ArrayList<Route> buildRoutes() {
        // declare variables
        ArrayList<Route> routes = new ArrayList<Route>();
        Route currentPair;
        Customer cust1, cust2;
        boolean c1, c2;

        // loop through every pair of customers
        for (int i = 0; i < _pairs.size(); i++) {
            // get customer from pair
            currentPair = _pairs.get(i);
            cust1 = currentPair.getFirstCustomer();
            cust2 = currentPair.getLastCustomer();
            // reset booleans for each customer
            c1 = false;
            c2 = false;
            // check if either customer from pair is already route
            for (int j = 0; j < routes.size(); j++) {
                if (routes.get(j).get_customerList().contains(cust1)) {
                    c1 = true;
                }

                if (routes.get(j).get_customerList().contains(cust2)) {
                    c2 = true;
                }
            }
            // if neither customer is route
            if (c1 == false && c2 == false) {
                // check that pair does not go over capacity
                if (cust1.c + cust2.c <= _prob.depot.c) {
                    // makes new route out of pair
                    Route newRoute = new Route();
                    newRoute.addToEnd(cust1);
                    newRoute.addToEnd(cust2);
                    routes.add(newRoute);
                    _pairs.remove(currentPair);
                    i--;
                }
                // if first customer not in route
            } else if (c1 == false) {
                for (Route route : routes) {
                    // check if any route ends in customer 2
                    if (route.getLastCustomer() == cust2) {
                        // check if merging would go over capacity
                        if ((route.get_requirment() + cust1.c) <= _prob.depot.c) {
                            // append customer to end of route
                            route.addToEnd(cust1);
                            _pairs.remove(currentPair);
                            i--;
                        }
                        // check if route starts with customer 2
                    } else if (route.getFirstCustomer() == cust2) {
                        // check if merging will not go over capacity
                        if ((route.get_requirment() + cust1.c) <= _prob.depot.c) {
                            // add customer to start
                            route.addToStart(cust1);
                            _pairs.remove(currentPair);
                            i--;
                        }
                    }
                }
                // if second customer is not in a route
            } else if (c2 == false) {
                for (Route route : routes) {
                    // check if any routes end with customer 1
                    if (route.getLastCustomer() == cust1) {
                        // check if merging will not go over capacity
                        if ((route.get_requirment() + cust2.c) <= _prob.depot.c) {
                            // append customer to end of route
                            route.addToEnd(cust2);
                            _pairs.remove(currentPair);
                            i--;
                        }
                        // check if any routes begin with customer 1
                    } else if (route.getFirstCustomer() == cust1) {
                        // check if merging will not go over capacity
                        if ((route.get_requirment() + cust2.c) <= _prob.depot.c) {
                            // append customer to start of route
                            route.addToStart(cust2);
                            _pairs.remove(currentPair);
                            i--;
                        }
                    }
                }
            }
        }
        return routes;
    }

}
```

References

CLARKE, G., AND WRIGHT, J. 1962. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research 12*, 4, 568–581.